Compiler construction

Lecture 5: Project extensions

Alex Gerdes
Vårtermin 2016

Chalmers University of Technology — Gothenburg University

This lecture

Some project extensions:

- Arrays
- Pointers and structures
- Object-oriented languages
- Module system

Arrays

Memory structure

JavaLette restrictions

- Only local variables and parameters – no global variables or other non-local declarations
- Only simple data types (`int`, `double`, `boolean`) with fixed-size values
- Only call-by-value parameter passing

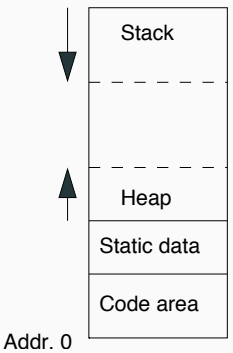
As a consequence, all data at runtime can be stored in the activation records on a stack.

More general language features

Global variables, nested procedures, linked structures and pointers, classes, ..., have other requirements: a stack is not sufficient for run-time memory management.

Memory organisation

Typical memory organisation



Comments

- Static area for global variables
- Stack and heap grow from opposite ends to avoid predetermined size decisions
- Stack managed by LLVM
- Management of heap much more complicated than stack

Arrays in JavaLette

Java-like arrays

Arrays in the JavaLette extensions are similar to Java arrays:

- A variable of type e.g. `int[]` contains a reference to a block of memory on the heap, where array elements are stored.
- Arrays must be explicitly created as in

```
int[] v = new int[20];
```

- Lower bound of array index is always `0`
- Array elements are initialized to `0`/`0.0`/`false`
- Arrays have a `length` attribute, with dot-notation
- Arrays can be both function arguments and results

Two extensions (each one credit)

First extension

One-dimensional arrays and 'foreach' statement, as in

```
int sum = 0;
for (int x : v)
  sum = sum + x;
```

The ordinary `for` statement is not required.

Two extensions (each one credit)

First extension

One-dimensional arrays and 'foreach' statement, as in

```
int sum = 0;
for (int x : v)
  sum = sum + x;
```

The ordinary `for` statement is not required.

Second extension

Multidimensional arrays:

- All indices must get upper bounds when an array is created
- For $n > 1$, an n-dimensional array is a one-dimensional array, each of whose elements is an $n - 1$-dimensional array

Sample LLVM code

Indexing in two-dimensional array

```
%arr1 = type %struct1*
%arr2 = type %struct2*

%struct1 = type {i32, [0 x i32]}
%struct2 = type {i32, [0 x %arr1]}

define i32 @getElem (%arr2 %m, i32 %i, i32 %j) {
  %p1 = getelementptr %arr2 %m, i32 0, i32 1, i32 %i
  %p2 = load %arr1* %p1
  %p3 = getelementptr %arr1 %p2, i32 0, i32 1, i32 %j
  %p4 = load i32* %p3
  ret i32 %p4
}
```

Your generated code may well look different.

Hints for array extension

First extension

- LLVM type of array hinted at in previous slide, use size `0`
- Use C function `calloc` to allocate `0`-initialized memory
- New forms of expression: array indexing and `new` expression
- Indexed expressions also as L-values in assignments
- Not required to generate bounds-checking code

Second extension

Hints for array extension

First extension

- LLVM type of array hinted at in previous slide, use size `0`
- Use C function `calloc` to allocate `0`-initialized memory
- New forms of expression: array indexing and `new` expression
- Indexed expressions also as L-values in assignments
- Not required to generate bounds-checking code

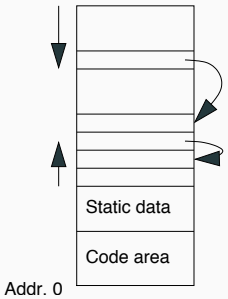
Second extension

- `new` expression with several indices involves generating code with loops and repeated `calloc`'s
- Indexing requires several `getelementptr` instructions

Structures and pointers

Structures and pointers extension

- In addition to function definitions, a Javalette file may contain definitions of structures and pointer types
- Structure objects are allocated on the heap, using `new`
- Pointer variable (on stack) may refer to memory structure on the heap



Structures and pointers examples

Code examples in JAVALETTE.

```
typedef struct Node *list;

struct Node {
  int elem;
  list next;
}

list cons(int x, list xs) {
  list res;
  res = new Node;
  res->elem = x;
  res->next = xs;
  return res;
}
```

```
int length(list xs) {
  if (xs == (list)null)
    return 0;
  else
    return 1 + length(xs->next);
}

list fromTo(int m, int n) {
 if (m > n)
  return (list)null;
 else
  return cons(m, fromTo(m + 1, n));
}
```

Adding structures and pointers to JAVALETTE

New toplevel definitions

- Structure definitions, examplified by `Node`
- Pointer type definitions, examplified by `list`

New expression forms

New statement forms

Adding structures and pointers to JAVALETTE

New toplevel definitions

- Structure definitions, examplified by `Node`
- Pointer type definitions, examplified by `list`

New expression forms

- Heap object creation, examplified by `new Node`
- Pointer dereferencing, examplified by `xs->next`
- Null pointers, examplified by `(list)null`

New statement forms

Adding structures and pointers to JAVALETTE

New toplevel definitions

- Structure definitions, examplified by `Node`
- Pointer type definitions, examplified by `list`

New expression forms

- Heap object creation, examplified by `new Node`
- Pointer dereferencing, examplified by `xs->next`
- Null pointers, examplified by `(list)null`

New statement forms

- Pointer dereferencing allowed in left hand sides of assignments, as in `xs->elem = 3;`
- In absense of garbage collection, you should have a `free` statement

Implementing structures and pointers in LLVM backend

Some hints

- Structure and pointer type definitions translate to LLVM type definitions
- Again, use `calloc` for allocating heap memory
- `getelementptr` and `load` will be used for pointer dereferencing
- Info about struct layout may be needed in the state of code generator

Implementing structures and pointers in LLVM backend

Some hints

- Structure and pointer type definitions translate to LLVM type definitions
- Again, use `calloc` for allocating heap memory
- `getelementptr` and `load` will be used for pointer dereferencing
- Info about struct layout may be needed in the state of code generator

From previous lecture: Computing the size of a type

We use the `getelementptr` instruction:

```
%p = getelementptr %T* null, i32 1
%s = ptrtoint %T* %p to i32
```

Now, `%s` holds the size of `%T`.

Other uses of pointers (not part of extension)

Code example (in C)

```
void swap (int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main () {
  int a = 1;
  int b = 3;
  swap(&a, &b);
  printf("a=%d\n", a);
}
```

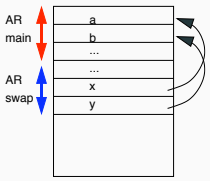
Other uses of pointers (not part of extension)

Code example (in C)

```
void swap (int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

int main () {
  int a = 1;
  int b = 3;
  swap(&a, &b);
  printf("a=%d\n", a);
}
```

Parameter passing by reference

- To make it possible to return results in parameters, one may use pointer parameters
- Actual arguments are addresses
- Problem: makes code optimization much more difficult



Call by reference and aliasing

Code examples

```
void swap (int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

...
swap (x, x);
...
```

Call by reference and aliasing

Code examples

```
void swap (int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}

...
swap (x, x);
...
```

Comments

- With call by reference and pointers, two different variables may refer to the same location; aliasing
- Aliasing complicates code optimization:

```
x := 2
y := 5
a := x + 3
```

Here we might want to replace last instr by `a := 5`; but what if `y` is an alias for `x`?

Deallocating heap memory

The problem

In contrast to stack memory, there is no simple way to say when heap allocated memory is not needed anymore.

Two main approaches

1. Explicit deallocation
   - Programmer deallocates memory when no longer needed (using `free`)
   - Potentially most efficient
   - Very easy to get wrong (memory leakage or premature returns)
2. Garbage collection
   - Programmer does nothing; runtime system reclaims unneeded memory
   - Secure but runtime penalty
   - Acceptable in most situations
   - Used in Java, Haskell, C#, ...

Garbage collection

General approach

Runtime system keeps list(s) of free heap memory, `malloc` returns a chunk from suitable free list.

Many variations. Some approaches:

1. Reference counting: each chunk keeps a reference count of incoming pointers, when count becomes zero, chunk is returned to free list; problem: cyclic structures
2. When free list is empty, collect in two phases:
   - **Mark** Follow pointers from global and local variables, marking reachable chunks
   - **Sweep** Traverse heap and return unmarked chunks to free list
3. Keep two heap areas: when active half becomes full, copy all reachable chunks to other half and continue with that half as active (free list not needed)

Object-orientation

Object-oriented languages

Class-based languages

We consider only languages where objects are created as instances of classes. A class describes:

- a collection of instance variables; each object gets its own copy of this collection
- a collection of methods to access and update the instance variables

Each object contains, in addition to the instance variables, a pointer to a class descriptor. This descriptor contains addresses of the code of methods.

Without inheritance, all this is straightforward; classes are just structures. We propose a little bit more: single inheritance without method override.

JavaLette classes, code example 1

```
class Counter {
  int val;

  void incr () {
    val++;
    return;
  }

  int value () {
    return val;
  }
}
```

```
int main () {
  Counter c;
  c = new Counter;
  c.incr();
  c.incr();
  c.incr();
  int x = c.value();
  printInt(x);
  return 0;
}
```

JavaLette classes, code example 2

```
class Point2 {
  int x;
  int y;

  void move(int dx, int dy) {
     x = x + dx;
     y = y + dy;
  }

  int getX() { return x; }

  int getY() { return y; }
}
```

```
class Point3 extends Point2 {
  int z;
  void moveZ(int dz) {
    z = z + dz;
  }
  int getZ() { return z; }
}

int main() {
  Point2 p;
  Point3 q = new Point3;

  q.move(2, 4);
  q.moveZ(7);
  p = q;
  ...
```

Adding classes to basic JavaLette (extension 1)

New toplevel definitions

- Class definitions, consisting of a number of instance variable declarations and a number of method definitions
- Instance variables are only visible within methods of the class
- All methods are public
- All classes have one default constructor, which initializes instance variables to default values (`0`, `false`, `null`)
- A class may extend another one, adding more instance variables and methods, but without overriding
- Classes are types; variables can be declared to be references to objects of a class
- We have subtyping; if `S extends C`, then `S` is a subtype of `C`; whenever an object of type `C` is expected, we may supply an object of type `S`.

Hints for object extension 1

New forms of expressions

- Object creation, examplified by `new Point2`, which allocates a new object on the heap with default values for instance variables
- Method calls, examplified by `p.move(3,5)`
- Null references, examplified by `(Point)null`
- Self reference. Within a class, `self` refers to the current object; all calls to sibling methods within a class must use method calls to `self`

Implementation hints

Much of ideas (and code) from the pointers/structures extension can be reused.

Method calls will be translated to function call with receiving object as extra, first parameter.

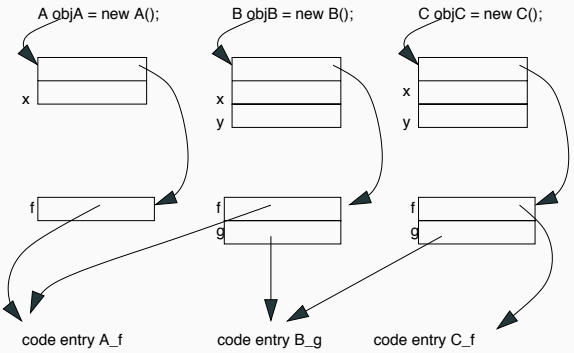
Overriding / dynamic dispatch (extension 2)

Example with overriding

We consider the following classes:

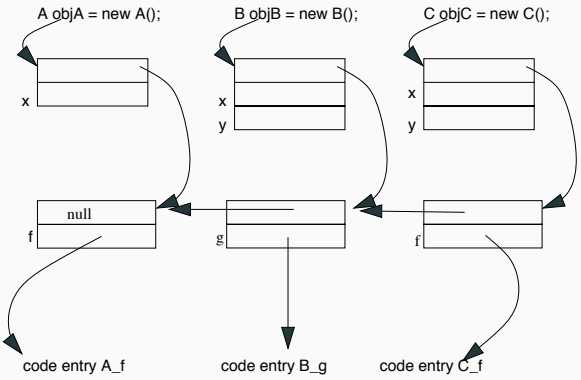
```
class A {
  int x;
  void f (int z) {x = z;}
}

class B extends A {
  int y;
  int g() {return 0;}
}

class C extends B {
  void f(int z) {x = z; y = 0;}
}
```

Objects at runtime



Each object includes instance variables and pointer to class descriptor.

Objects at runtime, variant



Class descriptors linked into list. List searched at runtime.

Dynamic dispatch

Code example

```
A obj = objA;
...
obj.f(5);
...
obj = objC;
...
obj.f(3);
```

Dynamic dispatch

Code example

```
A obj = objA;
...
obj.f(5);
...
obj = objC;
...
obj.f(3);
```

What code is run?

- The method to execute is determined at runtime by following the link to the class descriptor
- Static type checking guarantees that there is a method with proper signature in the descriptor
- There is an efficiency penalty in dynamic dispatch (so optimization tries to remove it)

Multiple inheritance

What is it?

The possibility of a class `C` to be a subclass of two (or more) unrelated classes `A` and `B`.

Problems

Uses

Disallowed in Java. (Note: Instead a Java class can implement multiple interfaces).

Allowed in C++. Indication of ambiguous method/attribute inheritance needed.

Multiple inheritance

What is it?

The possibility of a class `C` to be a subclass of two (or more) unrelated classes `A` and `B`.

Problems

- Cannot use the simple layout of objects and class descriptors shown before
- What if `A` and `B` have a method or attribute with the same name?

Uses

Disallowed in Java. (Note: Instead a Java class can implement multiple interfaces).

Allowed in C++. Indication of ambiguous method/attribute inheritance needed.

Modules

Module systems

Programmer's perspective: Modularity

- Reusability
- Information hiding
- Name space control

Module systems

Programmer's perspective: Modularity

- Reusability
- Information hiding
- Name space control

Compiler's perspective: Separate compilation

- Smaller compilation units
- Recompilation only of changed units
- Library modules released as binaries

Some approaches

Increasing levels of sophistication

- Inclusion mechanism: concatenate all files before compilation
- Include with header files: headers with type information included for compilation and separate linking
- Import mechanism:
  - Compilation requires interface info from imported files
  - Compilation generates interface and object files
  - Often in OO languages, module = class
- Language construct: module as separate form of data structure, independent of file structure

A possible module system for basic JavaLette

Extension proposal

- One module per file
- All modules in same directory (further extension: define search path mechanism)

Observations

- Mainly system for name space control and libraries
- If you want to implement it, you may get credits
- Difficulty: not much support in LLVM

Import in JavaLette

New syntax

If `M` is a module name, then

- `import M` is a new form of declaration
- `M.f(`e_1`, ..., `e_n`)` is a new form of expression

Unqualified use

A function in an imported module may be used without the module qualification if the name is unique. Name clashes are resolved as follows:

- If two imported modules define a function `f`, we must use the qualified form
- If the current module and an imported module both define `f`, the unqualified name refers to the local function

Import and dependency

Import

To use functions defined in `M`, another module must explicitly import `M`.

Hence, import is not transitive, i.e, if `M` imports `L` and `L` imports `K`, it does not follow that `M` imports `K`.

Dependency

- If `M` imports `L`, then `M` depends on `L`
- If `M` imports `L` and `L` depends on `K`, then `M` depends on `K`; dependency is transitive

We assume that dependency is non-cyclic: if `M` depends on `N`, then `N` may not depend on `M`.

Compiling a module, 1

Compiler's tasks

When called by `jlc M.jl`, the compiler must

1. Read the import statements of `M` to get list of imported modules
2. Recursively, read the import statements of these modules (and report an error if some module not found)
3. Build dependency graph of involved modules
4. Sort modules topologically (and report error if cyclic import found)
5. Go through modules in topological order (`M` last) and check timestamps to see if recompilation is necessary

Hint: It is OK to require that import statements are in the beginning of the file and with one import per line to avoid need of complete parsing.

Compiling a module, 2

Symbol table

You need a symbol table with types of functions from all imported modules. This info is readily available in LLVM files, but needs to be collected (and parsed).

Build the symbol table so that unqualified names will find the correct type signature (i.e., you must check for name clashes).

Note 1 It is a good idea to replace unqualified names by qualified (for code generation)

Note 2 Type declaration for all imported functions must be added to LLVM file